21 | CPU案例：如何提高LLC（最后一级缓存）的命中率?

2020-01-13 庄振运

讲述：庄振运
时长 13:36　大小 10.91M

你好，我是庄振运。

前面两讲中，我介绍了性能优化的六大原则和十大策略。从今天开始，我们来通过具体案例的解决方案讲解，了解这些原则和策略是如何应用的。

首先，我们要来探讨的是一个 CPU 相关的性能优化案例。

这个性能案例，是关于CPU 的最后一级缓存的。你应该知道，最后一级缓存（一般也就是 L3），如果命中率不高的话，对系统性能会有极坏的影响（相关基础知识建议回顾第 15 讲）。所以对这一问题，我们要及时准确地监测、暴露出来。

至于具体解决方案，我这里建议采取三种性能优化策略，来提高最后一级缓存的命中率。分别是：**紧凑化数据结构**、**软件预取数据**和**去除伪共享缓存**。它们分别适用于不同的情况。

性能问题：最后一级缓存（LLC）不命中率太高

一切问题的解决都要从性能问题开始入手，我们首先来看看**最后一级缓存不命中率太高**这个性能问题本身。

缓存的命中率，是 CPU 性能的一个关键性能指标。我们知道，CPU 里面有好几级缓存（Cache），每一级缓存都比后面一级缓存访问速度快。最后一级缓存叫 LLC（Last Level Cache）；LLC 的后面就是内存。

当 CPU 需要访问一块数据或者指令时，它会首先查看最靠近的一级缓存（L1）；如果数据存在，那么就是缓存命中（Cache Hit），否则就是不命中（Cache Miss），需要继续查询下一级缓存。

缓存不命中的比例对 CPU 的性能影响很大，尤其是最后一级缓存的不命中时，对性能的损害尤其严重。这个损害主要有两方面的性能影响：

第一个方面的影响很直白，就是 **CPU 的速度**受影响。我们前面讲过，内存的访问延迟，是 LLC 的延迟的很多倍（比如五倍）；所以 LLC 不命中对计算速度的影响可想而知。

第二个方面的影响就没有那么直白了，这方面是关于**内存带宽**。我们知道，如果 LLC 没有命中，那么就只能从内存里面去取了。LLC 不命中的计数，其实就是对内存访问的计数，因为 CPU 对内存的访问总是要经过 LLC，不会跳过 LLC 的。所以每一次 LLC 不命中，就会导致一次内存访问；反之也是成立的：每一次内存访问都是因为 LLC 没有命中。

更重要的是，我们知道，一个系统的内存带宽是有限制的，很有可能会成为性能瓶颈。从内存里取数据，就会占用内存带宽。因此，如果 LLC 不命中很高，那么对内存带宽的使用就会很大。内存带宽使用率很高的情况下，内存的存取延迟会急剧上升。更严重的是，最近几年计算机和互联网发展的趋势是，后台系统需要对越来越多的数据进行处理，因此**内存带宽越来越成为性能瓶颈**。

LLC 不命中率的测量

针对 LLC 不命中率高的问题，我们需要衡量一下问题的严重程度。在 Linux 系统里，可以用 Perf 这个工具来测量 LLC 的不命中率（在第 15 讲中提到过）。

那么 Perf 工具是怎么工作的呢?

它是在内部使用性能监视单元，也就是 PMU（Performance Monitoring Units）硬件，来收集各种相关 CPU 硬件事件的数据（例如缓存访问和缓存未命中），并且不会给系统带来太大开销。 这里需要你注意的是，PMU 硬件是针对每种处理器特别实现的，所以支持的事件集合以及具体事件原理，在处理器之间可能有所不同。

PMU 尤其可以监测 LLC 相关的指标数据，比如 LLC 读写计数、LLC 不命中计数、LLC 预先提取计数等指标。具体用 Perf 来测量 LLC 各种计数的命令格式是：

```
1 perf stat -e LLC-loads,LLC-load-misses,LLC-stores,LLC-store-misses
```

复制代码

下图显示的是一次 Perf 执行结果。

```
Performance counter stats for 'system wide':

    1,951,012,982      LLC-loads                                                   (57.54%)
      314,964,439      LLC-load-misses           #   16.14% of all LL-cache hits   (58.54%)
      313,740,538      LLC-stores                                                  (40.73%)
       78,295,790      LLC-store-misses                                            (39.07%)

     13.573636259 seconds time elapsed
```

我们可以看到，在这段取样时间内，有 1951M（19.51 亿）次 LLC 的读取，大约 16% 是不命中。有 313M（3.13 亿）次 LLC 的写入，差不多 24% 是不命中。

如何降低 LLC 的不命中率?

那么如何降低 LLC 的不命中率，也就是提高它的命中率呢？根据具体的问题，至少有三个解决方案。而且，这三个方案也不是互相排斥的，完全可以同时使用。

第一个方案，也是最直白的方案，就是**缩小数据结构**，让数据变得紧凑。

这样做的道理很简单，对一个系统而言，所有的缓存大小，包括最后一级缓存 LLC，都是固定的。如果每个数据变小，各级缓存自然就可以缓存更多条数据，也就可以提高缓存的命中率。这个方案很容易理解。

举个例子，开源的 C++ Folly 库里面有很多类，比如 F14ValueMap，就比一般的标准库实现小很多，从而占用比较少的内存；采用它的话，自然缓存的命中率就比较高。

第二个方案，是**用软件方式来预取数据**。

这个方案也就是通过合理预测，把以后可能要读取的数据提前取出，放到缓存里面，这样就可以减少缓存不命中率。“用软件方式来预取数据”理论上也算是一种“**用空间来换时间**”的策略（参见第 20 讲），因为付出的代价是占用了缓存空间。当然，这个预测的结果可能会不正确。

第三个方案，是具体为了解决一种特殊问题：就是伪共享缓存。伪共享缓存这个问题，我会在后面详细讲到。这个方案也算是一种“**空间换时间**”的策略，是通过让每个数据结构变大，牺牲一点存储空间，来解决伪共享缓存的问题。

除了最直白的缩小数据结构，另外两个解决方案（用软件方式来预取数据、去除伪共享缓存）都需要着重探讨。

软件提前预取指令

我们先展开讨论一下第二种方案，也就是用软件提前预取指令。

现代 CPU 其实一般都有**硬件指令**和**数据预取**功能，也就是根据程序的运行状态进行预测，并提前把指令和数据预取到缓存中。这种硬件预测针对连续性的内存访问特别有效。

但是在相当多的情况下，程序对内存的访问模式是随机、不规则的，也就是不连续的。硬件预取器对于这种随机的访问模式，根本无法做出正确的预测，这就需要使用**软件预取**。

软件预取就是这样一种预取到缓存中的技术，以便及时提供给 CPU，减少 CPU 停顿，从而降低缓存的不命中率，也就提高了 CPU 的使用效率。

现代 CPU 都提供相应的预取指令，具体来讲，Windows 下可以使用 VC++ 提供的 _mm_prefetch 函数，Linux 下可以使用 GCC 提供的 __builtin_prefetch 函数。GCC 提供了这样的接口，允许开发人员向编译器提供提示，从而帮助 GCC 为底层的编译处理器产生预取指令。这种策略在硬件预取不能正确、及时地预取数据时，极为有用。

但是软件预取也是有代价的。

一是预取的操作本身也是一种 CPU 指令，执行它就会占用 CPU 的周期。更重要的是，预取的内存数据总是会占用缓存空间。因为缓存空间很有限，这样可能会踢出其他的缓存的内容，从而造成被踢出内容的缓存不命中。如果预取的数据没有及时被用到，或者带来的好处不大，甚至小于带来的踢出其他缓存相对应的代价，那么软件预取就不会提升性能。

我自己在这方面的实践经验，有这么几条：

1. 软件预取最好只针对绝对必要的情况，就是对会实际严重导致 CPU 停顿的数据进行预取。
2. 对于很长的循环（就是循环次数比较多），尽量提前预取后面的两到三个循环所需要的数据。
3. 而对于短些的循环（循环次数比较少），可以试试在进入循环之前，就把数据提前预取到。

去除伪共享缓存

好了，我们接着来讨论第三个方案：去除伪共享缓存。

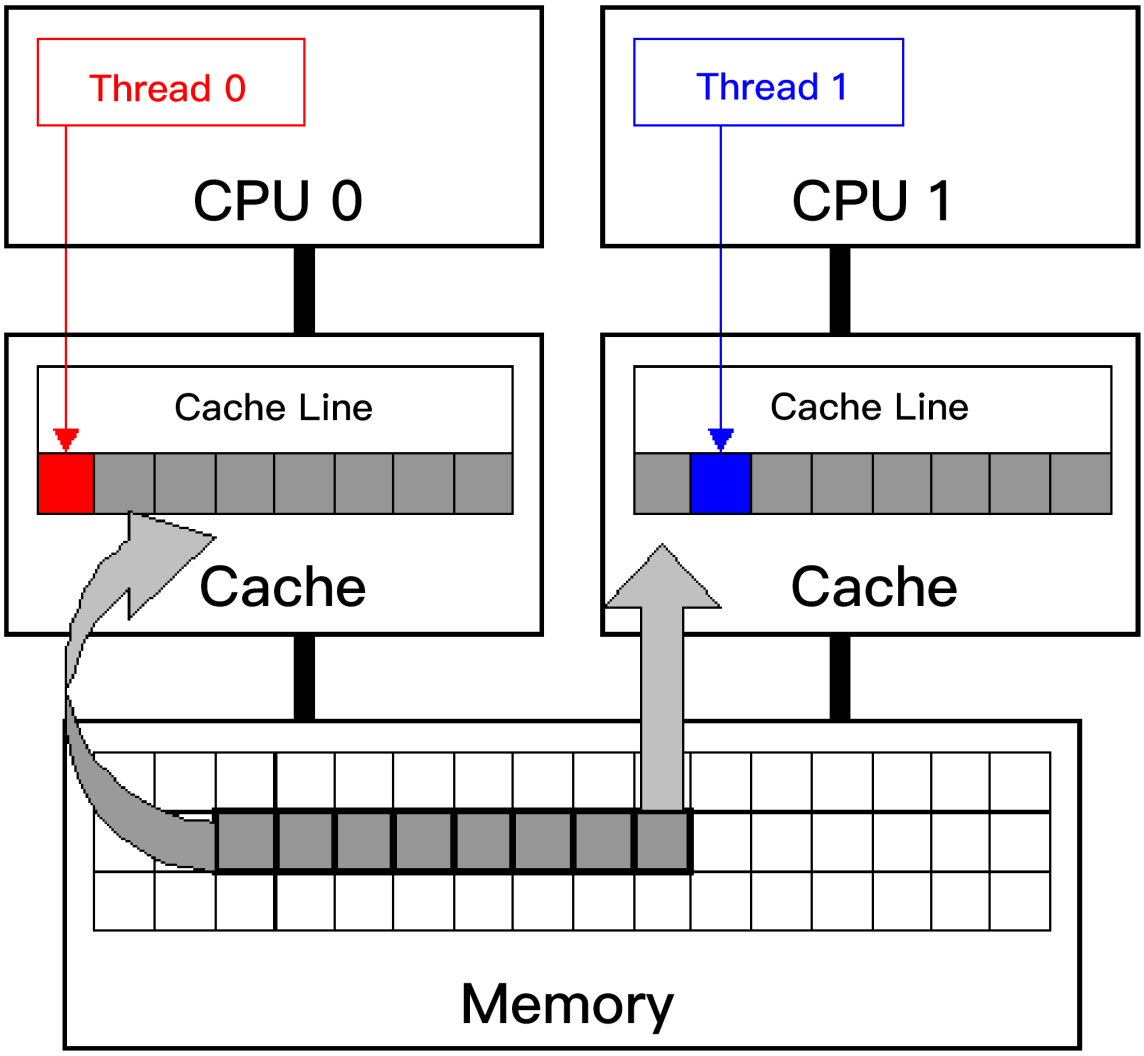
什么是伪共享缓存呢？

我们都知道，内存缓存系统中，一般是以缓存行（Cache Line）为单位存储的。最常见的缓存行大小是 64 个字节。现代 CPU 为了保证缓存相对于内存的一致性，必须实时监测每个核对缓存相对应的内存位置的修改。如果不同核所对应的缓存，其实是对应内存的同一个位置，那么对于这些缓存位置的修改，就必须轮流有序地执行，以保证内存一致性。

但是，这将导致核与核之间产生竞争关系，因为一个核对内存的修改，将导致另外的核在该处内存上的缓存失效。在多线程的场景下就会导致这样的问题。当多线程修改看似互相独立

的变量时，如果这些变量共享同一个缓存行，就会在无意中影响彼此的性能，这就是**伪共享**。

你可以参考下面这张 Intel 公司提供的图，两个线程运行在不同的核上，每个核都有自己单独的缓存，并且两个线程访问同一个缓存行。



如果线程 0 修改了缓存行的一部分，比如一个字节，那么为了保证缓存一致性，这个核上的整个缓存行的 64 字节，都必须写回到内存；这就导致其他核的对应缓存行失效。其他核的缓存就必须从内存读取最新的缓存行数据。这就造成了其他线程（比如线程 1）相对较大的停顿。

这个问题就是**伪共享缓存**。之所以称为“伪共享”，是因为，单单从程序代码上看，好像线程间没有冲突，可以完美共享内存，所以看不出什么问题。由于这种冲突性共享导致的问题

不是程序本意，而是由于底层缓存按块存取和缓存一致性的机制导致的，所以才称为“伪共享”。

我工作中也观察到好多次这样的伪共享缓存问题。经常会有产品组来找我们，说他们的产品吞吐量上不去，后来发现就是这方面的问题。所以，我们开发程序时，不同线程的数据要尽量放到不同的缓存行，避免多线程同时频繁地修改同一个缓存行。

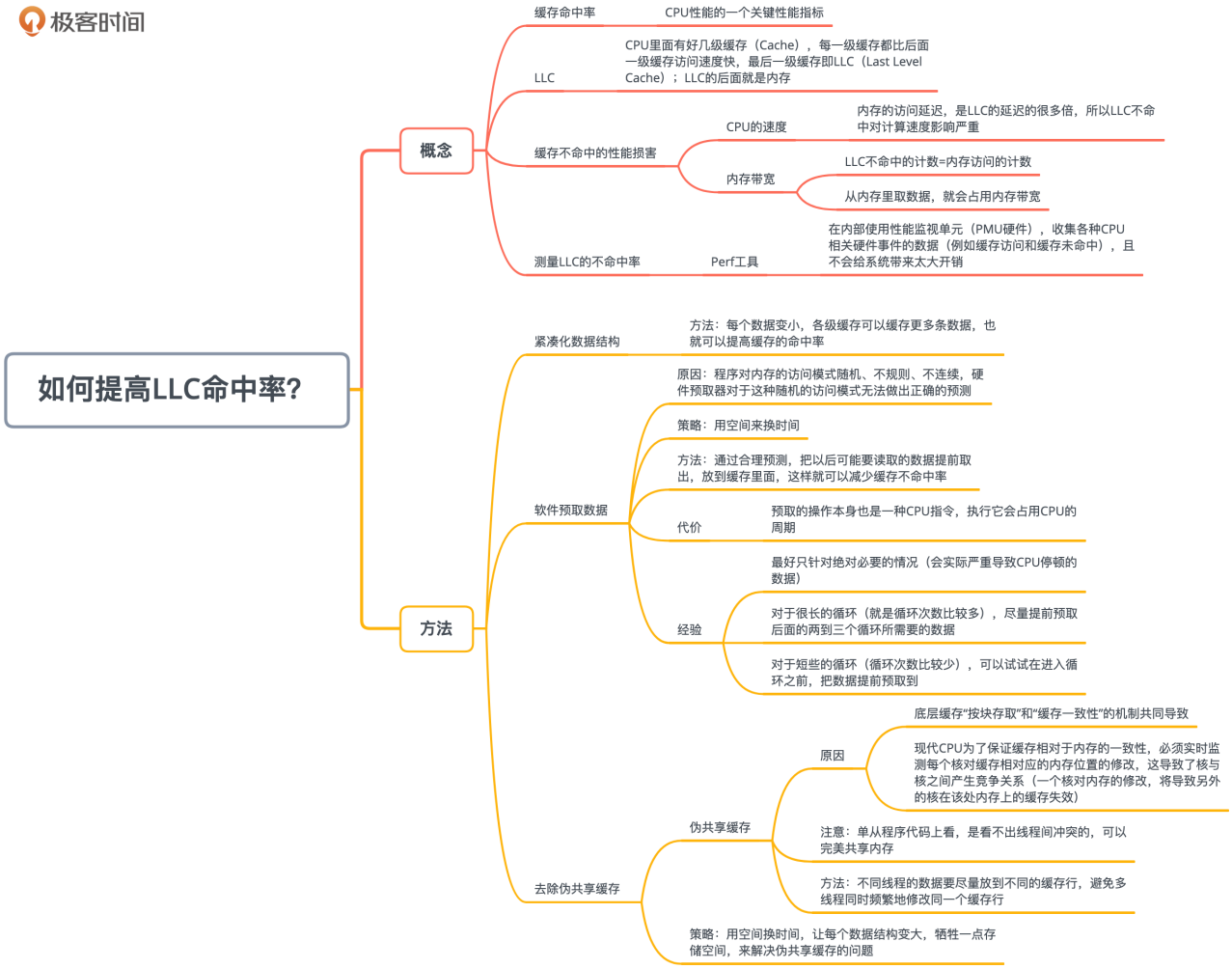
举个具体例子，假如我们要写一个多线程的程序来做分布式的统计工作，为了避免线程对于同一个变量的竞争，我们一般会定义一个数组，让每个线程修改其中一个元素。当需要总体统计信息时，再将所有元素相加得到结果。

但是，如果这个数组的元素是整数，因为一个整数只占用几个字节，那么一个 64 字节的缓存行会包含多个整数，就会导致几个线程共享一个缓存行，产生“伪共享”问题。

这个问题的解决方案，是**让每个元素单独占用一个缓存行**，比如 64 字节，也就是按缓存行的大小来对齐（Cache Line Alignment）。具体方法怎么实现呢？其实就是插入一些无用的字节（Padding）。这样的好处，是多个线程可以修改各自的元素和对应的缓存行，不会存在缓存行竞争，也就避免了“伪共享”问题。

总结

这一讲，我们介绍了 CPU 方面的优化案例，重点讨论了**如何降低 LLC 的缓存不命中率**。我们提出了三个方案，分别是紧凑化数据、软件指令预取和去除伪共享缓存。



尤其是第三个方案解决的伪共享缓存问题，对大多数程序员和运维人员而言，不太容易理解。为什么难理解？是因为它牵扯了软件（比如多线程）和硬件（比如缓存一致性和缓存行的大小）的交互。

当多线程共用同一个缓存行，并且各自频繁访问时，会导致严重的称为“伪共享”的性能问题。这种问题，恰如清代词人朱彝尊的两句词，“共眠一舸听秋雨，小簟轻衾各自寒”。所以需要我们狠狠心，把它们强行分开；“棒打鸳鸯”，让它们“大难临头各自飞”，其实呢，是为了它们都好。

思考题

硬件指令预取的基本原理是什么？为什么有时候非常有效，但有时候会有害呢？分别试举出一个具体的模块开发中的例子，来说明为什么有效和有害。

欢迎你在留言区分享自己的思考，与我和其他同学一起讨论，也欢迎你把文章分享给自己的朋友。

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇　20 | 性能优化十大策略：如何系统地有层次地优化性能问题?

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。